Zellic

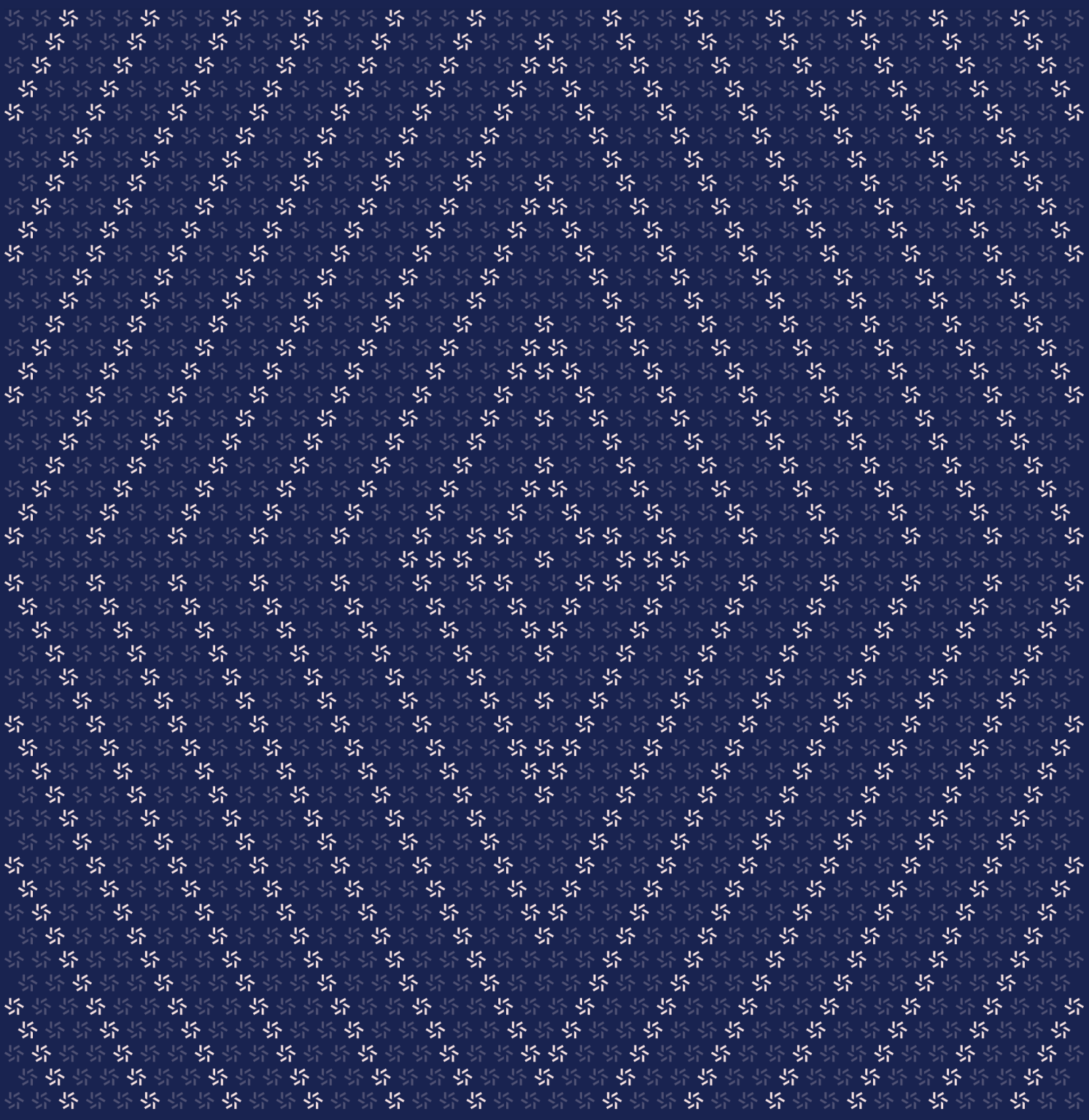
Prepared for
0xrelapse
Ferum Labs

Prepared by
Seunghyeon Kim
Jaeeu Kim
Zellic

February 29, 2024

Blackwing

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Ferum Labs from February 27th, 2024 to February 28th, 2024. During this engagement, Zellic reviewed Blackwing's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an attacker exploit an issue related to assets?
- How well is the deployer implemented within the system?
- Is there a potential for an attacker to create a balance mismatch with a vault token?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Deploy script of contracts
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

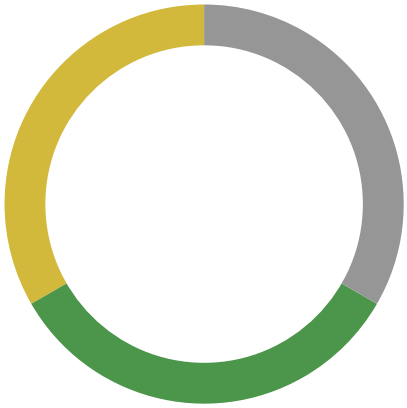
1.4. Results

During our assessment on the scoped Blackwing contracts, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Ferum Labs's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	1



2. Introduction

2.1. About Blackwing

Blackwing is a modular blockchain built for margin trading using a novel new primitive called Limitless Pools. Before launching, Blackwing is allowing users to deposit assets to earn XP and other partner incentives. To make sure user assets do not sit idle, the deposit contract can be configured to deploy assets using various backends. To start, USDC and ETH supplied will be deployed to Aave.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

> **Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.
>
> **Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.
>
> **Integration risks.** Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.
>
> **Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Blackwing Contracts

Repository	https://github.com/ferumlabs/evm-contracts ↗
Version	evm-contracts: `d5d0f772896f0c72c6179d3f946cf5b94cf4d53e`
Programs	• vault.sol • vault_token.sol • deployer/aave_deployer.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 0.3 person-weeks. The assessment was conducted over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Seunghyeon Kim
Engineer
seunghyeon@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 27, 2024	Kick-off call
February 27, 2024	Start of primary review period
February 28, 2024	End of primary review period

3. Detailed Findings

3.1. Return value checks for `transfer` and `transferFrom`

Target	BlackwingVault		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In the BlackwingVault contract, the `deposit()` and `withdraw()` functions do not include checks on the return values of `transfer` and `transferFrom`. This parameter should be validated to ensure the success of the transfer.

Impact

If the transfer of assets fails to revert, the vault may mint tokens without actually receiving the asset. This would increase the `totalSupply` of the vault token but would not add to the balance obtained from `getUndeployedAmount()`, as called within the `getTotalAssetAmount()` function.

```
function deposit(IERC20 asset, uint amount) public {
  requireAssetRegistered(asset);

  PoolInfo memory pool = pools[asset];
  uint totalVaultTokenSupply = pool.vaultToken.totalSupply();
  uint vaultTokensToMint = 0;
  if (totalVaultTokenSupply == 0) {
    vaultTokensToMint = amount * INITIAL_LIQUIDITY_MULTIPLIER;
  } else {
    uint totalAssetBalance = getTotalAssetAmount(asset);
    // ...
    vaultTokensToMint = amount * totalVaultTokenSupply / totalAssetBalance;
  }
  require(vaultTokensToMint > 0, VAULT_TOKEN_GRANULARITY_ERR);
  asset.transferFrom(msg.sender, address(this), amount);
  pool.vaultToken.mint(msg.sender, vaultTokensToMint);
  // ...
}

function getTotalAssetAmount(IERC20 asset) internal view returns (uint) {
  return getUndeployedAmount(asset) + getDeployedAmount(asset);
}
```

```
function getDeployedAmount(IERC20 asset) internal view returns (uint) {
  return pools[asset].deployer.totalDeployedAmount(asset);
}

function getUndeployedAmount(IERC20 asset) internal view returns (uint) {
  return asset.balanceOf(address(this));
}
```

Recommendations

We recommend using a wrapper such as SafeERC20 so that noncompliant ERC-20 tokens that do not return a boolean are safely handled.

For example, like this:

```
// ...
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/
    SafeERC20.sol";

contract BlackwingVault is Initializable, AccessControlUpgradeable {
 using SafeERC20 for IERC20;
  // ...

  function deposit(IERC20 asset, uint amount) public {
    // ...
   asset.transferFrom(msg.sender, address(this), amount);
   asset.safeTransferFrom(msg.sender, address(this), amount);
    // ...
  }

  function withdraw(IERC20 asset, uint vaultTokensToBurn) public {
    // ...
   asset.transfer(msg.sender, amountReturned);
   asset.safeTransfer(msg.sender, amountReturned);
    // ...
  }
```

Remediation

This issue has been acknowledged by Ferum Labs, and a fix was implemented in commit ce6b2a83 ↗.

3.2. No check that `pool[asset]` is registered

Target	BlackwingVault		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the BlackwingVault contract, the `updateDeployer()` function is responsible for modifying the deployer. It lacks a check that a pool is registered.

Impact

The pool of assets may not be registered at the time of the change. Currently, this does not pose an issue as all functions in the vault verify that the asset is registered within the pool. However, considering the upgradability of the contract and the potential addition of more functions, this could become a concern depending on the nature of future implementations.

Recommendations

We suggest implementing a check to verify the existence of the pool. This precautionary measure would prevent the protocol from setting the `deployer` on a pool that does not exist, adding an extra layer of security to the system.

```
function updateDeployer(IERC20 asset, IDeployer deployer) public {
  require(hasRole(OWNER_ROLE, msg.sender), UNAUTHORIZED_ERR);
 requireAssetRegistered(asset);
  pools[asset].deployer = deployer;
}
```

Remediation

This issue has been acknowledged by Ferum Labs, and a fix was implemented in commit d7158213 ↗.

The team added `requireAssetRegistered(asset)` for checking the pool.

3.3. Duplicate checks in deployer

Target	BlackwingAaveDeployer		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `BlackwingAaveDeployer` contract has two functions, `remove()` and `whitelistAsset()`, both of which have duplicate role checking.

- `remove` checks for `VAULT_ROLE` with a modifier and a require
- `whitelistAsset` checks for `OWNER_ROLE` with a modifier and a require

```
function whitelistAsset(IERC20 asset, IPoolAddressesProvider pap)
    public onlyRole(OWNER_ROLE) {
  require(hasRole(OWNER_ROLE, msg.sender), UNAUTHORIZED_ERR);
  // ...
}

function remove(IERC20 asset, uint amount) public onlyRole(VAULT_ROLE) {
  require(hasRole(VAULT_ROLE, msg.sender), UNAUTHORIZED_ERR);
  requireAssetWhitelisted(asset);
  // ...
}
```

Impact

The redundant checks consume unnecessary gas and diminish the code's readability and overall maturity.

Recommendations

Consider removing the duplicate checks to ensure the role is only checked a single time.

Remediation

This issue has been acknowledged by Ferum Labs, and a fix was implemented in commit d7158213 ↗.

The team removed `onlyRole(VAULT_ROLE)` and used `hasRole(VAULT_ROLE, msg.sender)`.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Centralization risk on the OWNER role

The `deployer` is responsible for receiving assets from the vault, supplying them to the Aave pool and transferring the `aToken` back to the vault. However, the owner role can modify the pool's `deployer` address at will.

```
function updateDeployer(IERC20 asset, IDeployer deployer) public {
  require(hasRole(OWNER_ROLE, msg.sender), UNAUTHORIZED_ERR);
  pools[asset].deployer = deployer;
}

function deployAssets(IERC20 asset, uint amount) public {
  require(hasRole(OWNER_ROLE, msg.sender), UNAUTHORIZED_ERR);
  requireAssetRegistered(asset);

  PoolInfo memory pool = pools[asset];
  require(asset.transfer(address(pool.deployer), amount),
    ASSET_DEPLOYMENT_ERR);
  pool.deployer.deploy(asset, amount);
}
```

The `deployer` in `deployAssets()` can be changed by the vault's owner, potentially allowing the owner to direct all vault balances to a deployer address that they control.

Blackwing has acknowledged this but maintains it for upgrading deploy code. Additionally, Blackwing plans to use a multi-sig wallet to ensure their `OWNER`'s security.

4.2. Front-run on initialize

Both BlackwingVault and BlackwingVaultToken utilize the `initialize()` function for contract initialization. The lack of an owner check could potentially expose it to front-running during deployment. However, the team is leveraging OpenZeppelin's Upgradeable contracts. This involves deploying the proxy contract and initializing it in the same transaction, providing protection against front-running attacks in this particular scenario.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: aave_deployer.sol

Function: `deploy(IERC20 asset, uint256 amount)`

This function deploys a pool with asset.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: Must be whitelisted.
  - **Impact**: Address of the asset which is supplied to aave pool.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount to be supplied to the pool.

Function: `registerVault(address vault)`

This function grants `VAULT_ROLE` to a given address.

Inputs

- `vault`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address to register.

Function: `remove(IERC20 asset, uint256 amount)`

This function withdraws the tokens from the pool.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: Must be whitelisted.
  - **Impact**: Address of the asset which is withdrew from aave pool.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount to be withdrawn.

Function: `whitelistAsset(IERC20 asset, IPoolAddressesProvider pap)`

This function adds asset to the pools.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Asset to whitelist.
- `pap`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: A pool-address provider.

5.2. Module: vault.sol

Function: `balance(IERC20 asset, address user)`

This function is used to return the asset amount corresponding to the vault token balance of the user.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: Only registered assets' address.
  - **Impact**: Address of the asset.
- `user`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Address of the user.

Branches and code coverage

Intended branches

- Return the asset amount corresponding to the vault token balance of the user.
  - ☑ Test coverage

Negative behavior

- Revert if asset is not registered.
  - ☑ Negative test

Function: `deployAssets(IERC20 asset, uint256 amount)`

This function facilitates the transfer of assets from the vault to the deployer; subsequently, the deployer is expected to call the supply function of the Aave pool. Consequently, the vault becomes the holder of the aToken from the Aave pool.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: Only registered assets' address.
  - **Impact**: Address of the asset.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of the asset to supply.

Branches and code coverage

Intended branches

- Transfer asset in the vault to deployer and call deployer.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test
- Revert if asset is not registered.
  - ☑ Negative test

- Revert if asset is not transferred to the deployer.
  - ☑ Negative test

Function: `deposit(IERC20 asset, uint256 amount)`

This function is used to deposit assets into the vault.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: Only registered assets can be used.
  - **Impact**: Address of the asset.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of the asset to deposit.

Branches and code coverage

Intended branches

- Calculate the amount of vault tokens to mint corresponding to the amount of asset deposited.
  - ☑ Test coverage
- Transfer asset from the sender to the vault.
  - ☑ Test coverage
- Mint vault tokens to the user.
  - ☑ Test coverage
- Update the last deposit block of the user.
  - ☑ Test coverage

Negative behavior

- Revert if asset is not registered.
  - ☑ Negative test
- Revert if asset is not transferred from the sender.
  - ☐ Negative test

Function: `disableLPTransfers(IERC20 asset)`

This function is used to disable transfers of the vault token of the asset.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: Only registered assets' address.
  - **Impact**: Address of the asset.

Branches and code coverage

Intended branches

- Update the vault token to disable transfers.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test
- Revert if asset is not registered.
  - ☑ Negative test

Function: `disableWithdrawals()`

This function is used to disable withdrawals from the vault.

Branches and code coverage

Intended branches

- Update the `withdrawsEnabled` to false.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test

Function: `enableLPTransfers(IERC20 asset)`

This function is used to enable transfers of the vault token of the asset.

Inputs

- `asset`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: Only registered assets' address.

Branches and code coverage

Intended branches

- Update the vault token to enable transfers.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test
- Revert if asset is not registered.
  - ☑ Negative test

Function: `enableWithdrawals()`

This function is used to enable withdrawals from the vault.

Branches and code coverage

Intended branches

- Update the `withdrawsEnabled` to true.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test

Function: `initialize(uint256 _minBlocksSinceLastDeposit)`

This function is used to initialize the contract.

Inputs

- `_minBlocksSinceLastDeposit`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the minimum number of blocks since last deposit.

Branches and code coverage

Intended branches

- Initialize the contract.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☐ Negative test

Function: `lastDepositBlock(address user)`

This function is used to return the last deposit block of the user.

Inputs

- `user`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the user.

Branches and code coverage

Intended branches

- Return the last deposit block of the user.
  - ☑ Test coverage

Function: `registerAsset(IERC20 asset, BlackwingVaultToken vaultToken, IDeployer deployer)`

This function is used to register a pool of asset in the vault.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the asset.
- `vaultToken`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: Address of the vault token.

- `deployer`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deployer.

Branches and code coverage

Intended branches

- Update the pool info with provided asset, vault token, and deployer.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test
- Reverts if target pool is already registered.
  - ☑ Negative test

Function: `removeAssets(IERC20 asset, uint256 amount)`

This function is used to transfer aToken to the deployer and invoke the remove function of the deployer, with the remove function anticipated to call the withdraw function of the Aave pool. This process enables the vault to retrieve the asset from the Aave pool.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: Only registered assets' address.
  - **Impact**: Address of the asset.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of the asset to withdraw.

Branches and code coverage

Intended branches

- Transfer aToken in the vault to deployer and call remove.

☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  ☑ Negative test
- Revert if asset is not registered.
  ☑ Negative test
- Revert if asset is not transferred to the deployer.
  ☑ Negative test

Function: `setMinBlocksSinceLastDeposit(uint256 _minBlocksSinceLastDeposit)`

This function is used to set the minimum number of blocks since last deposit.

Inputs

- `_minBlocksSinceLastDeposit`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the minimum number of blocks since last deposit.

Branches and code coverage

Intended branches

- Update the `minBlocksSinceLastDeposit` to the provided value.
  ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  ☑ Negative test

Function: `updateDeployer(IERC20 asset, IDeployer deployer)`

This function is used to update the deployer address of the pool of asset.

Inputs

- `asset`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: Address of the asset.

- `deployer`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the deployer.

Branches and code coverage

Intended branches

- Update the deployer address of the pool of asset.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `withdraw(IERC20 asset, uint256 vaultTokensToBurn)`

This function is used to withdraw assets from the vault.

Inputs

- `asset`
  - **Control**: Arbitrary.
  - **Constraints**: Only registered assets' address.
  - **Impact**: Address of the asset.
- `vaultTokensToBurn`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of the vault tokens to burn.

Branches and code coverage

Intended branches

- Calculate the amount of asset to return to the user corresponding to the vault tokens to burn.
  - ☑ Test coverage
- Transfer the vault tokens to the vault.
  - ☑ Test coverage

- Burn the vault tokens of the user.
  - ☑ Test coverage

Negative behavior

- Revert if asset is not registered.
  - ☑ Negative test
- Revert if withdraws are disabled.
  - ☑ Negative test
- Revert if the user has not deposited for a minimum number of blocks.
  - ☑ Negative test
- Revert if asset is not transferred to the user.
  - ☐ Negative test

5.3. Module: vault_token.sol

Function: `disableTransfers()`

This function disables the `transfer` and `transferFrom` functions.

Function: `enableTransfers()`

This function enables the `transfer` and `transferFrom` functions.

Function: `initialize(address _vault, string name, string symbol)`

This function initializes the contract.

Inputs

- `_vault`
  - **Control**: None. Initialize will be called by deploy script.
  - **Constraints**: None.
  - **Impact**: Vault address to be.
- `name`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Name of vault token.
- `symbol`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Symbol of vault token.

Function: `mint(address to, uint256 amount)`

This function mints the token.

Inputs

- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address to get the tokens.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount to mint the token.

Branches and code coverage

Intended branches

- The function must mint the token to given `to`.
  - ☐ Coverage test

Negative behavior

- Revert if the function caller does not have `VAULT_ROLE`.
  - ☐ Negative test

Function: `transferFrom(address sender, address recipient, uint256 amount)`

This function transfers the vault token from a given `sender`.

Inputs

- `sender`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of token sender.
- `recipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of token recipient.
- `amount`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: Amount of token.

Function: `transfer(address recipient, uint256 amount)`

This function transfers the vault token.

Inputs

- `recipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address to get transferred token.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of token to transfer.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Blackwing contracts, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was informational in nature. Ferum Labs acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.